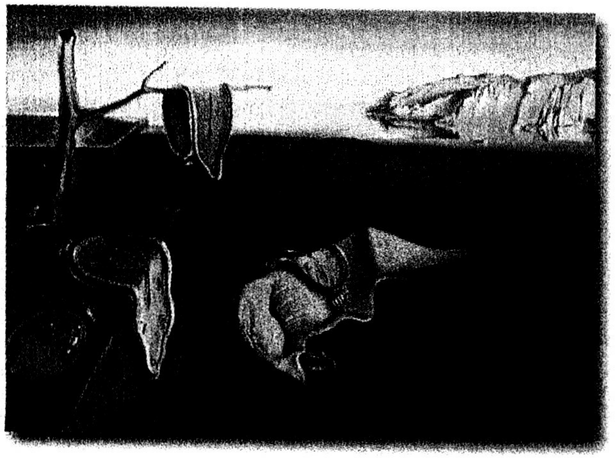
Poetry in programs: A brief examination of software aesthetics, including observations on the history of programming styles and speculations on post-object programming

Robert Filman
RIACS
NASA Ames
Moffett Field, CA



Lisp Poems

- One of the first projects Dan told me he planned was to create a book of *Lisp Poems*

```
((lambda (x)
   (list x (list 'quote x)))
 '(lambda (x)
    (list x (list 'quote x))))
```

Second Poem

```
((lambda (y)
      (letrec
          ((rev
            (lambda (x)
              (cond ((null? x) ())
                    (#t (append (rev (cdr x))
                                (list (if (pair? (car x))
                                          (rev (car x))
                                          (car x)))))))))
        (list (list (rev y) 'quote) (rev y))))
   '(lambda (y)
       (letrec
           ((rev
             (lambda (x)
               (cond ((null? x) ())
                     (#t (append (rev (cdr x))
                                 (list (if (pair? (car x))
                                           (rev (car x))
                                           (car x)))))))))
         (list (list (rev y) 'quote) (rev y)))))
```

Second Poem Eval

```
((((((y rev)
        ('quote (y rev) list)
        list)
       (((((((((x car)
                ((x car) rev)
                ((x car) pair?)
                if)
               list)
              ((x cdr) rev) append) #t)
            (() (x null?)) cond)
           (x) lambda) rev))
       letrec) (y) lambda) quote)
    ((((y rev)
       ('quote (y rev) list) list)
      (((((((((x car)
               ((x car) rev)
               ((x car) pair?)
               if)
             list)
            ((x cdr) rev) append) #t)
          (() (x null?))
          cond)
         (x) lambda) rev))
      letrec) (y) lambda))
```

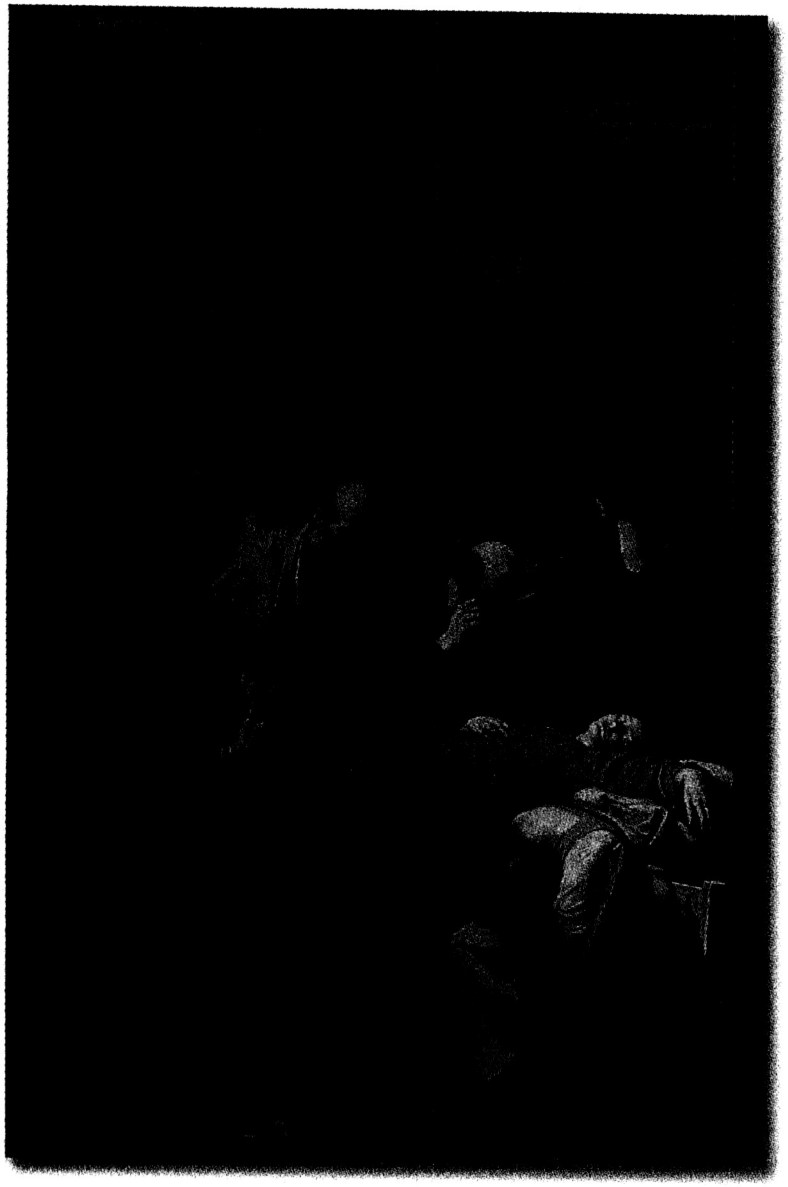
Third Poem (D.R.H.)

```
((lambda (y)
   (letrec
       ((rev
         (lambda (x)
           (cond ((null? x) ())
                  (#t (append
                       (rev (cdr x))
                       (list (cond
                              ((pair? (car x)) (rev (car x)))
                              ((symbol? (car x))
                               (string->symbol
                                (list->string
                                 (rev
                                  (string->list
                                   (symbol->string (car x)))))))
                              (#t (car x))))))))))
     (list (list (rev y) 'quote) (rev y))))
 '(lambda (y)
    (letrec
        ((rev
          (lambda (x)
            (cond ((null? x) ())
                   (#t (append
                        (rev (cdr x))
                        (list (cond
                               ((pair? (car x)) (rev (car x)))
                               ((symbol? (car x))
                                (string->symbol
                                 (list->string
                                  (rev
                                   (string->list
                                    (symbol->string (car x)))))))
                               (#t (car x))))))))))
      (list (list (rev y) 'quote) (rev y)))))
```

Third Poem Eval

```
 ((((((y ver)
    ((etouq etouq)
     (y ver) tsil) tsil)
   ((((((((((x rac) #t)
           (((((((x rac) gnirts>-lobmys) tsil>-gnirts) ver)
              gnirts>-tsil) lobmys>-gnirts)
            ((x rac) ?lobmys))
           (((x rac) ver)
            ((x rac) ?riap)) dnoc) tsil)
         ((x rdc) ver) dneppa) #t)
       (() (x ?llun)) dnoc) (x) adbmal) ver))
   certel) (y) adbmal) quote)
((((y ver)
   ((etouq etouq)
    (y ver) tsil) tsil)
  ((((((((((x rac) #t)
          (((((((x rac) gnirts>-lobmys) tsil>-gnirts) ver)
             gnirts>-tsil) lobmys>-gnirts)
           ((x rac) ?lobmys))
          (((x rac) ver)
           ((x rac) ?riap)) dnoc) tsil)
        ((x rdc) ver) dneppa) #t)
      (() (x ?llun)) dnoc) (x) adbmal) ver))
  certel) (y) adbmal))
```

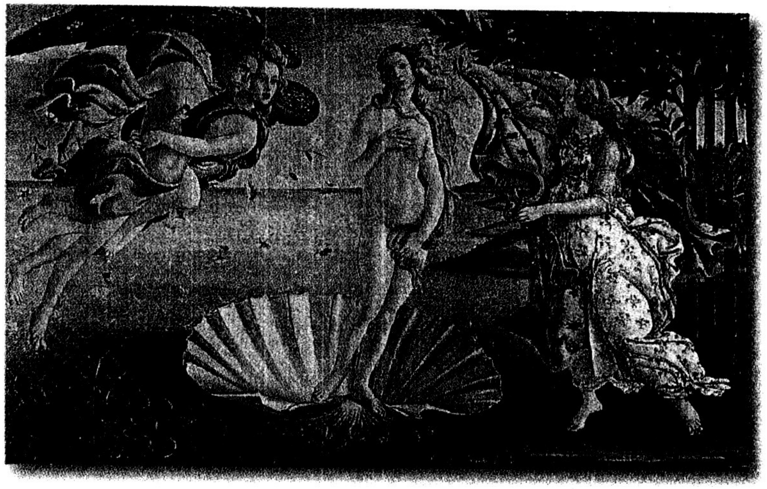
Fourth Poem



```
((lambda (x y)
   (list y
    (list 'quote x)
    (list 'quote y)))
 '(lambda (x y)
    (list y
     (list 'quote x)
     (list 'quote y)))
 '(lambda (x y)
    (list x
     (list 'quote x)
     (list 'quote y))))
```

The Art of Computer Programming (D.E.K.)

- Software development is an Art
- Art:
  - – Skill at joining or fitting.
  - – A system of principles and rules for attaining a desired end
  - – Use of skill to create that which is esthetically or intellectually pleasing
  - – Necromancy



Intellectual activities

- Science: Distillation of knowledge into principles and laws
- Engineering: The combination of art with attention to economy
- Manufacturing: Repeated activity following a well-defined and low-skill plan
- Fashion: Selecting from equivalent alternatives

Progress

- Arts, sciences, engineering show an intellectual progression, shaped by
  - New technology
  - Shifting economic forces
  - New understandings
  - Evolving responses to the ideas of prior generations

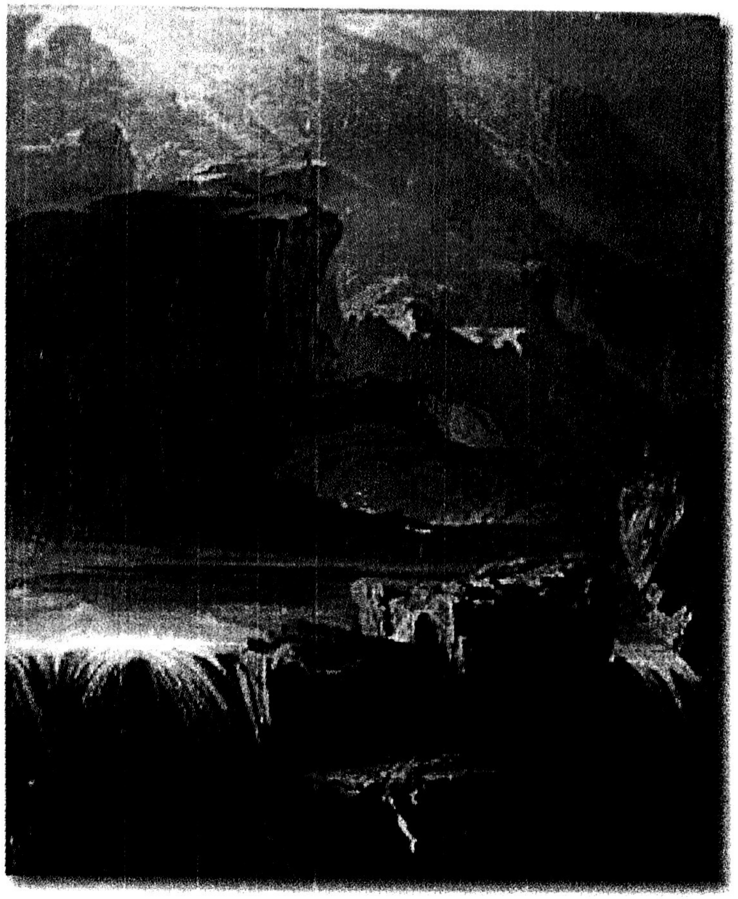
- Primitive
- Greek & Roman
- Byzantine
- Romanesque & Gothic
- Renaissance
- Baroque & Rococo
- Neoclassicism & Romanticism
- Impressionism
- Modern
- Post-modern

Monotonicity (or lack thereof)

- Science and engineering are unconditionally monotonic
  - No going back to Newtonian physics, Geometry = Euclid, Linnaeus
- Fine arts revisit old themes with new twists
  - Photorealism
- Disciplines like education and business management follow fashions

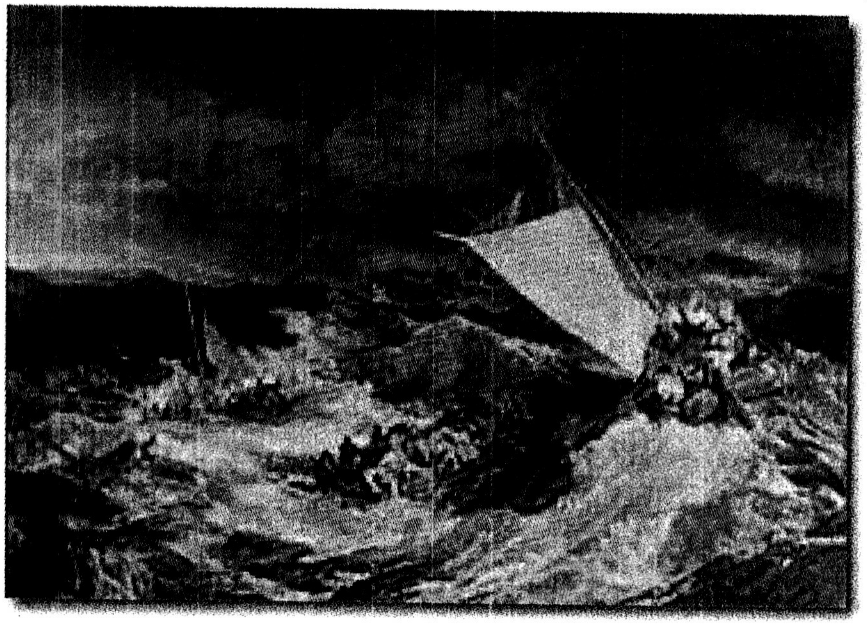
The Ilities of Software Development

- The joy of computer science is that it spans so much of the human skill set, from science to engineering to psychology
- Ilities
  - Aesthetic of understandability
  - Ease of
    - Construction
    - Maintenance
    - Evolvability
  - Economy of execution
  - Reliability
  - Security
  - Interoperability
  - …



Sapir-Whorf hypothesis applied to software development

- The programming language you use affects the way you think about software development
  - Half the gang-of-four patterns are patterns only because their addressing C++ programmers, not Lispers.



Programming Languages as an Intellectual Progression

- Programming is specification (M.W.)
- Earliest programming languages were concerned with "efficient realism"
  - Difficult to render even highly structured problems into code
  - Efficient use of machine resources was a dominent criterion
- Programming was *linear*
  - Things said in a program had a "one-to-one" correspondence to what happened in execution
- Programming was *planar*
  - One could easily trace the potential execution paths of a program and identify which conditions would give rise to which code being executed

Programming Language Eras

- Pure functionality
- Structured programming
- Abstract data types
- Object-oriented programming

- Functional programming
- Logic programming
- Rule-based systems

Limits of object-orientation

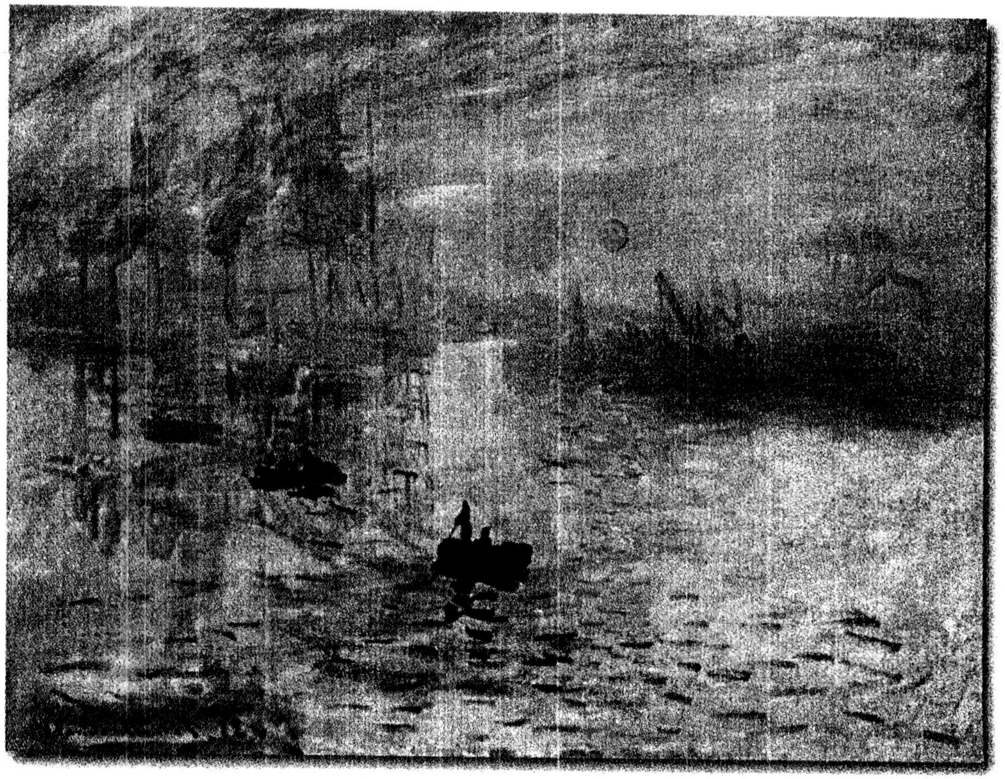
- All meaning is wrapped up in the code
- Unitary modularization
  - Tyranny of the dominant decomposition (H.O.)
- The world isn’t made up of discrete, unconnected objects
- Inherent inability to create and maintain correct code
- Tyranny of call-response
- Domain independence

Possible responses to the limitations

- All meaning is wrapped up in the code
  - Richer uses of annotation
    - Executable annotation, not UML or comments
- Unitary modularization
  - Aspect-oriented programming
- The world isn’t made up of discrete, unconnected objects
  - Composites, collections and masses
  - Maintained relationships
  - Persistence
  - More of a merger of the database notions of view and search with programming structures

Possible responses to the limitations, cont.

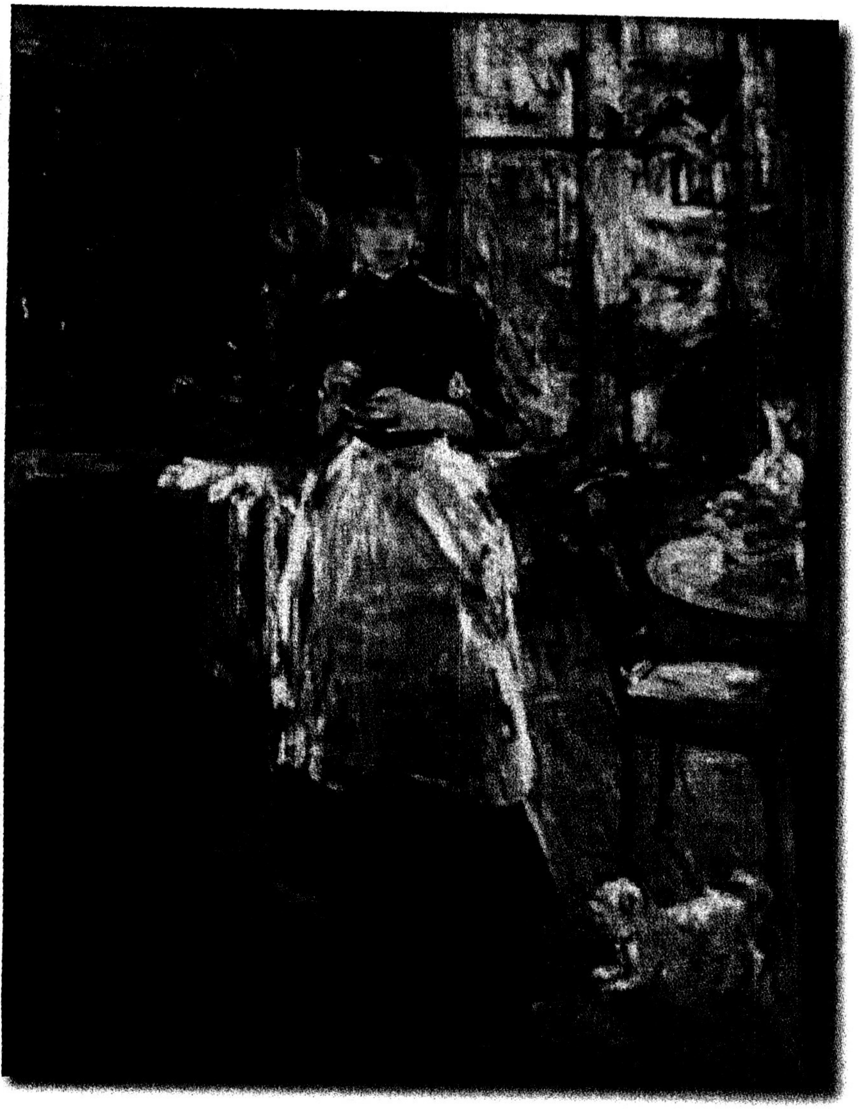
- Inherent inability to create and maintain correct code
  - Autonomic computing
    - Describe how to recognize incorrect behavior and what to do about it
- Tyranny of call-response
  - Event-based computing
  - Conversations, protocols
  - Context-aware systems
- Domain independence
  - Domain-specific languages
  - Extensible syntax



Concerns

- Programmers have many concerns—things they care about—when building software systems
- Current programming technology demands a dominant decomposition
  - Programmers have to program to all their concerns
    - Even the ones that don’t exist yet
  - Programmers have to know when to invoke other behavior
- Separation of concerns in conventional languages
  - Subprograms
  - Inheritance

Examples of Concerns



- Security
- Accounting
- Synchronization
- Quality of service
- Reliability
- Performance enhancements

- Concerns exist at both the requirements and design levels

Aspect-Oriented Programming

- Allows the separate specification of concerns
- Describes how concerns interact with the overall system and each other (annotation)
- Provides a tool that weaves together the separate concerns into a complete system

Aspect-Oriented Programming is Quantification and Obliviousness (R.E.F. & D.P.F.)

- The essence of the AOP idea is to allow
  - Write statements about part of or the entire program (quantification)
  - Where individual program elements don't have any notation that the alternative concerns are going to be invoked (obliviousness)

Trinity (R.E.F., K.H. & D.H.)

- Quantification over what?
  - The syntactic structure of the program
  - The result of static semantic (compiler) analysis
  - Events that happen dynamically in the course of program execution
- Sometimes there is a strong correspondence between syntactic structures, semantic objects and dynamic events
  - Sometimes there's not
- The shadow of a quantification is the places in the code that might affect the quantification

Trinity behavior

- Transform programs based on pattern-action rules
  - When the pattern of a quantification is seen, transform the program to perform the behavior desired in the action
  - Rules like database queries
- Transformations can be either
  - Structural: change the original program
  - Behavioral: perform some action before, after, around or instead of an original target
- Structural changes on events don’t make sense

Applications

- Debugging
- Profiling
- Monitoring
- Contextual evaluation (the "jumping beans" problem)
- Autonomic computing
- Security
- Concurrency
- Resource management

- Refactoring
- Persistence
- User interface consistency

Discussion